下载APP

08 | 单元测试应该怎么写？

2021-08-20 郑晔

《程序员的测试课》 课程介绍 >

讲述：郑晔
时长 11:22 大小 10.41M

你好，我是郑晔！

经过前面的介绍，我们已经对测试的基础有了理解，已经会用自动化测试框架来写测试了。对于那些不可控的组件，我们也可以用 Mock 框架将其替换掉，让测试环境变得可控。其实，我们在前面介绍的这些东西都是为了让我们能够更好地编写单元测试。

单元测试是所有测试类型中最基础的，它的优点是运行速度快，可以尽早地发现问题。只有通过单元测试保证了每个组件的正确性，我们才拥有了构建系统的一块块稳定的基石

按道理来说，我们应该尽可能多地编写单元测试，这可以帮助我们提高代码质量以及更准确地定位问题。但在实际的工作中，真正大面积编写单元测试的团队却并不多。前面我们

已经提到了一部分原因（比如设计没有做好），也有团队虽然写了单元测试，但单元测试没有很好地起到保护网的作用，反而是在代码调整过程中成了阻碍。

这一讲，我们就把前面学到的知识串联起来，谈谈如何做好单元测试。

单元测试什么时候写

你是怎么编写单元测试的呢？很多人的做法是先把所有的功能代码都写完，然后，再针对写好的代码一点一点地补写测试。

在这种编写测试的做法中，单元测试扮演着非常不受人待见的角色。你的整个功能代码都写完了，再去写测试就成了一件为了应付差事不得不做的事情。更关键的一点是，你编写的这些代码可能是你几天的工作量，你已经很难记得在编写这堆代码时所有的细节了，这个时候补写的测试对提升代码质量的帮助已经不是很大了。

所以，想要写好单元测试，最后补测试的做法总是很糟糕的，仅仅比不写测试好一点。你要想写好单元测试的话，**最好能够将代码和测试一起写**。

你或许会说，我在功能写完后立即就补测试了，这不就是代码和测试一起写的吗？其中的差异在于，把所有的功能写完的这个粒度实在是太大了。为一个大任务编写测试，是一件难度非常大的事，这也是很多人觉得测试难写的重要因素。要想做好单元测试，关键就是工作的粒度要小。

如果你学过《10x 程序员工作法》，或许已经听出来了。没错，这里的关键点就是要做好任务分解，而任务分解的关键就是粒度要小。

> I’m not a great programmer; I’m just a good programmer with great habits.
>
> 我不是一个伟大的程序员，只是一个有着好习惯的优秀程序员。
>
> —— Kent Beck

任务分解是每个程序员都应该拥有的好习惯，即便你**想写好单元测试也要从任务分解开始**。所以，你需要把一个要完成的需求拆分成很多颗粒度很小的任务。粒度要小到可以在很短时间内完成，比如，半个小时就可以写完。只有能够把任务分解成微操作，我们才能

够认清有足够的心力思考其中的每个细节。千万不要高估自己对于任务把控的粒度，**一定要把任务分解到非常小，这是能够写好代码，写好测试的前提条件，甚至可以说是最关键的因素**（如何具体分解一个需求，我曾经在《10x 程序员工作法》中专门用了一讲的篇幅进行介绍，如果你有兴趣不妨去回顾一下）。

当我们把需求拆分成颗粒度很小的任务时，我们才开始进入到编码的状态。而从这里开始，我们进入到代码和测试一起写的状态。

编写单元测试的过程

对于一个具体的任务，我们首先要弄清楚的是，怎么样算是完成了。一个完整的需求我们需要知道其验收标准是什么。具体到一个任务，虽然没有业务人员给我们提供验收标准，我们自己也要有一个验收标准，我们要能够去衡量怎么样才算是这个代码写合格了。

经过我们这一系列关于测试的介绍，你应该已经知道我要说什么了：**一个任务的代码要通过测试才算编码阶段的完成。**

但测试用例从哪来呢？这就需要我们设计了。不同于业务测试的测试用例，我们现在要写的是单元测试。而我们要测的单元现在还没有写，所以，没有人会给我们提供测试用例，单元测试的用例只能我们自己来。

还记得我们在实战里怎么做的添加 Todo 项吗？接下来，我们就结合这个部分来谈谈具体怎么做。

我们首先要确定的是待测单元的行为，也就是要实现的类里的一个函数，它的行为是什么样的。或许你已经发现了，这其实就是一个软件设计的过程。这里的设计指的是微观的设计，就是具体的一个函数准备写成什么样子。通常到了动手写代码这一步，大的设计已经在前面做完了。

因为我们现在不仅仅要写代码，还要写测试。所以，我们在设计这个函数接口时，还必须增加一点考量：它要怎么测。

在添加一个 Todo 项时，我们经过设计出来的函数接口就是下面这样。

```
1 TodoItem addTodoItem(final TodoParameter todoParameter);
```

复制代码

有了一个具体的函数接口设计，我们就可以针对它进行更具体的测试用例设计，也就是设计测试用例来描述这个接口的行为。

是的，这里我们并没有着急写代码。对很多人来说，写代码的优先级很高，但是，如果不在这里停一下的话，你可能就不会去思考是否还有要考虑的问题，而是直奔代码细节去了。而当我们专注于细节时，有限的注意力就会让你忽略掉很多东西。所以，**先设计测试用例，后写代码**，这是一个编码习惯的问题。

有了添加 Todo 项接口之后，我们就准备了两个测试场景：

添加正常的参数对象，返回一个创建好的 Todo 项；

添加空的参数对象，抛出异常。

有了测试场景，接下来把这些场景实例化出来，这个步骤相对来说就比较简单了。比如，对于添加正常的参数对象来说，那什么样的参数对象是正常的？我们就代入一个具体的正常参数（比如 foo）。有了这个实例化过的参数，我们就可以把具体的测试用例表现出来了。

复制代码

```
1  @Test
2  public void should_add_todo_item() {
3      TodoItemRepository repository = mock(TodoItemRepository.class);
4      when(repository.save(any())).then(returnsFirstArg());
5      TodoItemService service = new TodoItemService(repository);
6
7      TodoItem item = service.addTodoItem(new TodoParameter("foo"));
8
9      assertThat(item.getContent()).isEqualTo("foo");
10 }
```

在实际的工作中，究竟是先写测试，还是先写实现代码，这是个人工作习惯的问题。当我们有了测试用例之后，其实就是把一个具体的任务进一步拆分成更小的子任务了。**只要我们完成一个子任务，我们就可以做一次代码的提交，因为我们这个时候，既有测试代码又有实现代码，而且实现代码是通过了测试的。**

测接口还是测实现？

不知道你是否注意到了，在前面我一直在说，我们要测的是函数接口的行为。我一直说，单元测试是一种白盒测试。在一些人的理解中，白盒测试的关注点应该是内部实现。那单元测试到底应该关注接口，还是应该关注实现呢？

或许你还不清楚二者之间的区别，让我们把前面添加 Todo 项的例子拿过来。如果采用更加面向实现的做法，我们应该对 addTodoItem 这个函数的内部实现有进一步的约束，就像下面这样。

复制代码

```
@Test
public void should_add_todo_item() {
    TodoItemRepository repository = mock(TodoItemRepository.class);
    when(repository.save(any())).then(returnsFirstArg());
    TodoItemService service = new TodoItemService(repository);

    TodoItem item = service.addTodoItem(new TodoParameter("foo"));

    assertThat(item.getContent()).isEqualTo("foo");
    verify(repository).save(any());
}
```

这段代码中核心的差别就是增加了一句 verify，这也就意味着，我规定在 addTodoItem 的实现中必须要调用 repository 的 save 函数。

你或许会好奇，repository 本来就要调用 save 方法，那我在这里校验它调用了 save 方法，似乎也没什么大不了的。

单独这么看确实看不出什么问题，但是，如果你有很多测试都是这么写，当你准备重构时，你就会发现问题了。很多团队代码一调整，测试就失败，一个重要的原因就是代码实现和测试之间紧紧地绑定在了一起。因为测试约束的是实现细节，而只要调整实现细节，测试当然就失败了。这也是很多团队抱怨单元测试问题很多的重要原因。

所以，**在实际的项目中，我会更倾向于测试接口，尽可能减少对于实现细节的约束。**其实，这个原则不仅仅是在接口层面上，在一些测试的细节上也可以这么约定，比如下面这行代码。

复制代码

```
when(repository.save(any())).then(returnsFirstArg());
```

这其实是一种宽泛的写法，所以用了 any。如果严格限制的话，应该严格限定一个非常具体的参数。

复制代码

```
when(repository.save(new TodoItem("foo"))).then(returnsFirstArg());
```

同样，上一讲我们讲到了 Moco，我们设置模拟服务器可以设置得非常具体，像下面这样。

复制代码

```
server
  .request(and(by("foo"), by(uri("/foo"))))
  .response(and(with(text("bar")), status(200)));
```

也可以设置得非常宽泛，像这样。

复制代码

```
server.request(by(uri("/foo"))).response("bar");
```

除非这个测试里面有多个类似的请求，必须要做区分，否则，我倾向于使用宽泛一些的约束。这在某种程度上会降低未来重构代码时带来的影响。

不过实话说，要想完全消除对于实现细节的依赖，有时候也是很难的。比如在我们前面的 TodoItemService 的例子里面，repository 本身也是 TodoItemService 的一种实现细节，一旦进行一些重构，把 repository 的依赖从 TodoItemService 中拿掉，很多测试代码也需要调整。所以，在实际的项目中，我们只能说尽可能减少对于实现细节的依赖。

其实，关于实现细节的测试也是一种重复，等于你用测试把代码又重新写了一遍。程序员的工作中有一种重要的原则：DRY（Don’t Repeat Yourself），这不仅仅是说代码中不

要有重复，而且各种信息都不要重复（我在《软件设计之美》中讲过 DRY 原则，有兴趣不妨回顾一下）。

我建议你在设计单元测试的时候不要面向实现细节。但反过来，有些时候测试确实会漏掉一些细节，尤其是一些实现代码中的分支。怎么样发现自己的代码中是否有遗漏呢？这就是我们下一讲要讲的内容：测试覆盖率。

总结时刻

今天我们讲了如何去写单元测试。很多团队由于多方面的原因（比如设计做得不好），导致单元测试写得少。但为了提高代码质量以及更准确地定位问题，我们应该多写单元测试。

单元测试最好是和实现代码一起写，以便减少后续补测试的痛苦。想写好测试，关键要做好任务分解，否则，面对一个巨大的需求，没有人知道如何去给它写单元测试。

编写单元测试的过程，实际上就是一个任务开发的过程。一个任务代码的完成，不仅仅是写了实现代码，还要通过相应的测试。一般而言，任务开发要先设计相应的接口，确定其行为，然后根据这个接口设计相应的测试用例，最后，把这些用例实例化成一个个具体的单元测试。

单元测试常见的一个问题是代码一重构，单元测试就崩溃。这很大程度上是由于测试对实现细节的依赖过于紧密。一般来说，单元测试最好是面向接口行为来设计，因为这是一个更宽泛的要求。其实，在测试中的很多细节也可以考虑设置得宽泛一些，比如模拟对象的设置、模拟服务器的设置等等。

如果今天的内容你只能记住一件事，那请记住：**做好任务分解，写好单元测试。**

思考题

今天我们讨论了如何写好单元测试，你在实际项目中写过单元测试吗？你遇到了哪些问题，或者有哪些经验可以分享呢？欢迎在留言区分享你的观点。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | Mock 框架：怎么让测试变得可控？

下一篇 09 | 测试覆盖率：如何找出没有测试到的代码？

精选留言 (3)

写留言

Alexdown
2021-08-20

有时候写完单元测试，对于关键的“单元”可能还需要看看其运行时间以及内存分配情况（基准/性能测试）。请问老师，程序员是否要进行性能测试？如何进行性能测试？可否加餐一篇聊聊？谢谢

1

X
2021-08-23

请问老师，如果 接口行为Mock，入参设置宽泛一些（any），那这样入参感觉没有测试到，细节上会不会有啥问题？ 现在测试都是严格限制出入参。。

展开

作者回复: 这取决于你要测的是什么，我在专栏里建议的是测试接口，在这种情况下，Mock 的入参就不是关键点，为啥要限制那么严格呢？如果限制严格了，就是在做细节的测试。

大碗
2021-08-20

请问老师这个addItem的verify可以去掉么，不写测试覆盖度不会下降

作者回复: 不会，verify是一种断言，不会影响覆盖率。